UNIVERSIDAD DE LA RIOJA

TRABAJO FIN DE ESTUDIOS

Título

Refactorización de código heredado en el ámbito de la Gestión Segura del Espacio

Autor/es

PABLO MENDOZA GARCÍA

Director/es

JUAN FÉLIX SAN JUAN DÍAZ y ROSARIO LÓPEZ GÓMEZ

Facultad

Facultad de Ciencia y Tecnología

Titulación

Grado en Ingeniería Informática

Departamento

MATEMÁTICAS Y COMPUTACIÓN

Curso académico

2019-20

publicaciones.unirioja.es
E-mail: publicaciones@unirioja.es

UNIVERSIDAD
DE LA RIOJA

Facultad de Ciencia y Tecnología

Trabajo Fin de Grado

Grado en Ingeniería Informática

Refactorización de código heredado en el ámbito de la Gestión Segura del Espacio

Realizado por:
Pablo Mendoza García

Tutelado por:
Dra. Rosario López Gómez
Dr. Juan Félix San Juán Díaz

Logroño, Junio de 2020

Resumen

Al principio de la era espacial, los centros dependientes de los gobiernos fueron los principales encargados de crear, mantener y operar los distintos sistemas y dispositivos de vigilancia del espacio que rodea a nuestro planeta. Con el paso de los años, se han implementado diversas aplicaciones informáticas que dan soporte a este complejo sistema. Es importante destacar que gran parte de este software fue desarrollado utilizando técnicas y herramientas ahora en desuso. En este proyecto se aborda el problema de refactorización de un propagador orbital como un primer paso para su adaptación a las necesidades actuales y futuras de algunos de los procesos implicados en los sistemas de Gestión Segura del Espacio.

In the beginning of the space age, the goverment dependent space administrations were the main developers, maintainers and operators of various systems for the observation of near-Earth space. As the years passed by, multiple software applications have been developed to work with these complex systems. These systems were often developed using obsolete techiques and architectures. This project focuses on making changes to an orbital propagation software, as a fisrt step to adaptation to the current and future needs of the Space Situational Awareness systems.

Agradecimientos

Quiero agradecer el apoyo prestado durante la realización de este TFG a los tutores, por haberme dado la oportunidad de trabajar sobre un problema real e interesante, su apoyo para hacerme entender la teoría y el código, y por haber estado siempre ahí para resolver mis dudas, y llevarme en la dirección adecuada.

Igualmente, quiero agradecer a todos mis amigos, a Ramón, Pablo, Daniel, Álex, Alba, Noe y al resto de los *Pitukis* y amigos del *Pisito del Amor*, que estando presentes de uno u otro modo durante la cuarentena, hayan conseguido que los últimos cuatro meses se hayan soportado mejor, y a todos los que me han dado ánimos en mi última etapa de la carrera.

Agradezco también a mis padres, porque han estado ahí siempre que lo he necesitado.

Por último, quiero agradecer a Mariola, esa persona tan especial que todos los días me saca una sonrisa, me ha dado el ánimo y la confianza en mí mismo necesarios para continuar con mi último semestre y este trabajo, en un momento difícil para ambos.

Índice general

1. Descripción del Proyecto **6**
- 1.1. Solución semi-analítica . . . 6
- 1.2. Proyecto. Alcance del TFG . . . 7
  - 1.2.1. Puesta en marcha del propagador USM. . . . 8
  - 1.2.2. Refactorización del propagador USM. . . . 9
- 1.3. Planificación del TFG . . . 9
  - 1.3.1. Problemas y planes de contingencia . . . 9
  - 1.3.2. Planificación . . . 10
- 1.4. Tecnologías . . . 12

2. 1ª iteración: Puesta en marcha del propagador USM **14**
- 2.1. Análisis del código . . . 14
- 2.2. Compilación y puesta en marcha . . . 18
- 2.3. Validación . . . 21
- 2.4. Descomposición de tareas para la 2ª iteración . . . 21

3. 2ª iteración: Refactorización del propagador USM **24**
- 3.1. Refactorización . . . 24
  - 3.1.1. Documentación . . . 24
  - 3.1.2. Actuación general . . . 25
  - 3.1.3. Solucionar los problemas detectados . . . 28
- 3.2. Análisis de eficiencia . . . 31
  - 3.2.1. Análisis de velocidad de ejecución . . . 31
  - 3.2.2. Análisis de uso de memoria . . . 32
- 3.3. Resultado de la refactorización . . . 33

4. Conclusiones **37**
- 4.1. Conocimientos adquiridos . . . 38
- 4.2. Desviaciones temporales . . . 38

A. Elementos orbitales **41**

B. Reuniones **42**

Introducción y motivación del proyecto

Un propagador orbital implementa en un lenguaje de programación una solución de las ecuaciones diferenciales que describe el movimiento de cualquier objeto, tanto natural como artificial, que orbita alrededor de nuestro planeta, RSO (por su nombre en inglés, Resident Space Object). De este modo, los propagadores orbitales permiten calcular la posición y velocidad de un RSO en cualquier instante de tiempo a partir de una posición y velocidad conocidas. Los propagadores orbitales se utilizan principalmente en algunas fases del análisis y diseño de misiones espaciales y en numerosos procesos de la gestión segura del espacio, más conocida por su denominación en inglés Space Situational Awareness (SSA), entre ellos destacaremos:

- Mantenimiento de catálogos de basura espacial.
- Evitar colisiones en el espacio.
- Reentrada de RSO en la Tierra.
- ...

Tradicionalmente, los métodos de integración que proporcionan las soluciones que se implementan en los propagadores orbitales son de tres tipos: numéricos, analíticos y semi-analíticos. Los métodos numéricos permiten considerar modelos de fuerzas muy precisos, pero por el contrario su eficiencia depende del paso de integración. Los analíticos son eficientes desde el punto de vista computacional pero no son precisos porque sólo incluyen modelos simplificados de perturbaciones. Finalmente, los semi-analíticos combinan la precisión de los métodos numéricos con la eficiencia de los analíticos.

El Universal Semi-Analytical Model (USM) es un propagador orbital semi-analítico que fue desarrollado a comienzos de 1980 en la extinta Unión Soviética por el Dr. Yurasov [6]. La USM fue utilizada en diversos procesos del programa SSA de la URSS como son la reentrada de la estación espacial Salyut-7, los satélites

Cosmos o los orbitadores FSW-1-5 lanzados por China. El propagador USM fue proporcionado al Grupo de Computación Científica por el Dr. Paul J. Cefola [2]. Este propagador formó parte de un acuerdo de colaboración entre organizaciones estadounidenses (Naval Research Laboratory, Charles Stark Draper Laboratory y Massachusetts Institute of Technology) y rusas (Academy of Sciences), en el ámbito del SSA.

Aunque, la tecnología informática con la que se desarrolló este propagador ha quedado obsoleta y, por tanto, puede considerarse código heredado, los algoritmos que implementa siguen estando vigentes. Es por esta razón y de acuerdo con el Dr. Cefola que hemos decidido abordar un proyecto para su recuperación y adaptación a las nuevas necesidades del sector espacial.

Este TFG consta de cuatro capítulos. En el capítulo primero se introduce el proyecto en el que se enmarca esta memoria así como las tareas específicas y las tecnologías que se han empleado durante el desarrollo de este TFG. La puesta en marcha del propagador USM se describe en el capítulo segundo. El problema de la refactorización, es el tema que se aborda en el capítulo tercero. Finalmente, el capítulo cuarto está dedicado a las conclusiones.

Capítulo 1

Descripción del Proyecto

En esta sección se describe brevemente la teoría matemática que implementa el propagador orbital semi-analítico USM. A continuación, se presenta el proyecto de recuperación y actualización del propagador USM definido por los Dres. Cefola y San Juan. Se hace especial hincapié en las tareas específicas que se abordarán en este TFG. Para finalizar, se enumeran las tecnologías que han sido empleadas a lo largo del desarrollo de este TFG.

1.1. Solución semi-analítica

La solución semi-analítica que implementa el propagador orbital está formulada en variables no singulares $(a, \xi, \eta, P, Q, \lambda)$. La definición de estas variables en función de los elementos orbitales $(a, e, i, \omega, \Omega, M)$ es:

$$\begin{aligned} a &= a, \\ \xi &= e\cos(\omega+\Omega), \\ \eta &= e\sin(\omega+\Omega), \\ P &= \sin(i/2)\cos\Omega, \\ Q &= \sin(i/2)\sin\Omega, \\ \lambda &= M+\omega+\Omega. \end{aligned}$$

En el apéndice A se proporciona una descripción más detallada de los elementos orbitales.

La tabla 1.1 muestra las perturbaciones incluidas en el propagador USM. La primera columna muestra las fuerzas que perturban el movimiento del RSO: potencial terrestre, efectos producidos por el Sol y la Luna, presión de radiación solar y frenaje atmosférico. En la segunda, se describe la contribución de cada una de estas fuerzas a las ecuaciones de trasformación, estos términos se denominan de corto periodo, mientras que en la tercera, la contribución a las ecuaciones

Perturbaciones	**Términos de corto periodo**	**Ecuaciones promediadas**
Armónico zonal de segundo grado C_{20}	Términos de primer orden en a, ξ, η, P, Q, λ Términos de segundo orden en a	Términos de primer y segundo orden relativos a C_{20}
Armónicos lm del Geopotencial $(2 < l < 70, 0 < m < 70)$	Términos lineales	Términos lineares incluyendo resonancias
Atracción de la Luna y del Sol	Términos lineales	Términos lineales
Frenaje atmosférico (modelo de densidad GOST 25645.115-84)	No	Términos lineales y acoplados con c_{20}
Presión de radiacción solar	No	Términos lineales

Tabla 1.1: Perturbaciones incluidas en el propagador semi-analítico USM.

promediadas. Los términos de corto periodo, aquellos cuya frecuencia es inferior a un periodo orbital, son eliminados de las ecuaciones promediadas utilizando el método generalizado de promedios. Las ecuaciones promediadas sólo contienen los términos de largo periodo, lo cual permite emplear pasos de integración grandes y hace que el propagador semi-analítico sea muy eficiente desde el punto de vista computacional.

Una descripción más detallada de los modelos de fuerzas empleados en el propagador USM y del método generalizado de promedios se encuentra en las referencias [5] y [1]. Estas referencias han sido claves para entender el modelo de fuerzas y cómo fue implementado este propagador orbital.

1.2. Proyecto. Alcance del TFG

El proyecto que se va a abordar en este TFG se enmarca en el ámbito del código heredado, ya que la tecnología que se empleó en el desarrollo del propagador USM ha quedado obsoleta. El código que fue proporcionado al Dr. San Juan formaba parte de una librería DLL, la cual fue implementada con una de las primeras versiones del compilador Borland C++. Sin embargo, los algoritmos y parte de su diseño continúan siendo válidos y podrían reutilizarse en un nuevo propagador que heredase los más de veinte años de desarrollo del propagador USM. Esto permitirá iniciar un proyecto a partir de una aplicación ampliamente testeada y validada. Es importante mencionar que junto al código inicial se nos proporcionaron los resultados gráficos de las comparaciones realizadas por el Dr. Cefola y su equipo entre el propagador USM y la versión de la Draper Semi-Analitical Satellite Theory (DSST) incluida en el Goddard Trajectory Determination System (GTDS) [2].

El proyecto se va a desarrollar empleando una metodología iterativa, que abordará desde el problema de su puesta en marcha hasta su integración con DACE

(Differential Algebra Core Engine) [3], y SPICE[1]. Este nuevo propagador se convertirá en una herramienta muy útil para los técnicos e investigadores que mantienen e investigan en los sistemas de SSA (España participa en el programa de SSA de la Unión Europea).

DACE es una librería que implementa el cálculo de incertidumbres utilizando álgebra diferencial, está desarrollada en C mientras que utiliza C++ para su interfaz. Por otro lado, Spice es una librería desarrollada por la NASA en varios lenguajes de programación, entre ellos C; proporciona funcionalidades como cambios de sistemas de referencia, efemérides para los cuerpos del sistema solar del Jet Propulsion Laboratory (JPL),...

Este proyecto se va a desarrollar en cuatro iteraciones:

1. Puesta en marcha del propagador USM.
2. Refactorización del propagador USM.
3. Integrar DACE y emplear los kernel de Spice en el propagador USM.
4. Mejora del modelo de perturbaciones y de los métodos numéricos que se emplean en el propagador USM.

Debido al alcance de este proyecto, en este TFG sólo se van a abordar las dos primeras iteraciones. Las tareas específicas que se llevarán a cabo en cada una de estas dos iteraciones se describen a continuación.

1.2.1. Puesta en marcha del propagador USM.

Los objetivos que se persiguen en esta iteración son los siguientes: comprender el diseño del propagador orbital y su equivalencia con los algoritmos teóricos implementados, identificar los puntos de actuación para su refactorización, teniendo en cuanta su integración con DACE y Spice, y disponer de una aplicación totalmente funcional que permita crear test unitarios y test de integración para las siguientes iteraciones. En esta primera iteración se van a abordar las siguientes tareas:

1. Análisis del código.
2. Extraer el código de la librería DLL y eliminar las ligaduras del código del compilador Borland.
3. Compilar y ejecutar el código; para ello se va a implementar las funciones que faltan, se eliminaran los errores y se analizaran los avisos (warnings) del código.

[1] https://naif.jpl.nasa.gov/naif/toolkit.html

1.2.2. Refactorización del propagador USM.

El objetivo que se persigue en esta iteración consiste en adaptar el diseño y el código del propagador USM para su futura integración con las librerías DACE y Spice. Las tareas que se realizaran en esta iteración son:

1. Abordar los problemas detectados en la primera iteración.
2. Generar una documentación automática del código con Doxygen.
3. Realizar test unitarios y de integración.
4. Se analizará el efecto de la refactorización en la eficiencia del código.

1.3. Planificación del TFG

En esta sección se describen los problemas que podemos encontrarnos durante el desarrollo de este TFG junto con las acciones que nos permitirán solucionarlos. Por último, se muestra la planificación inicial a través de un diagrama de Gantt.

1.3.1. Problemas y planes de contingencia

Este TFG forma parte de un proyecto más amplio, que pretende proponer soluciones reales en la gestión de un sistema de SSA, se prevé que surjan problemas de muy variada índole, como son los que se describen a continuación:

- Falta de experiencia en proyectos que implican código heredado. Tecnología y prácticas de programación obsoletas. Código incompleto y mal documentado.
- Escasa formación para abordar un proyecto que implica fuertes conocimientos de dinámica orbital, matemáticas y física.
- Falta de experiencia para desarrollar un sistema robusto de validación y testeo en proyectos que implican un conocimiento profundo de la aritmética del ordenador.

En resumen, la mayoría de los problemas que se han previsto están relacionados con los conocimientos en los que se circunscribe este proyecto. En este sentido, la figura de los tutores será fundamental para proporcionar los recursos formativos necesarios. En cuanto a los problemas relacionados con las tecnologías, además de los tutores y la experiencia adquirida en la titulación , disponemos de numerosos recursos documentales e Internet.

1.3.2. Planificación

En el siguiente diagrama de Gantt (figura 1.1) se muestra la planificación temporal que inicialmente se ha previsto para este TFG:

Tareas	Inicio	Final	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10
Estudio teórico	03/02/20	09/02/20										
Análisis de código	10/02/20	23/02/20										
Extracción de la librería DLL	24/02/20	15/03/20										
Desligar el código a Borland	16/03/20	05/04/20										
Conseguir una versión funcional	30/03/20	12/04/20										
Tareas	Inicio	Final	S11	S12	S13	S14	S15	S16	S17	S18	S19	S20
Resolver problemas detectados en análisis	13/04/20	03/05/20										
Generar documentación	04/05/20	10/05/20										
Realizar test unitarios e integración	11/05/20	31/05/20										
Analizar eficiencia de la versión final	01/06/20	07/06/20										
Elaboración de la memoria	08/06/20	21/06/20										

Figura 1.1: Planificación temporal.

1.4. Tecnologías

En esta subsección se describen las tecnologías utilizadas para la realización de este TFG.

GNU GCC. El proyecto se ha desarrollado utilizando el compilador `g++ 10.1.0` (código abierto).

Assert. Esta librería se ha utilizado para realizar los test unitarios.

JetBrains CLion. Es un entorno de desarrollo de C/C++. Este IDE ha sido seleccionado por su sistema de búsqueda, autocompletado e inspección de código (utilizado bajo licencia de estudiante).

Valgrind. Es una suite de código abierto para la depuración de problemas de memoria y rendimiento de programas. Las herramientas utilizadas han sido útiles para:

- Comprobar los errores típicos del manejo de memoria dinámica en la transición hacia la memoria estática. Detectar problemas de inicialización de variables utilizadas sin inicializar, por lo que ha sido de gran ayuda en la sanitización del código.
- Para comprobar las llamadas en tiempo de ejecución y que se mantiene la lógica del propagador entre las iteraciones. La visualización se ha realizado utilizando KDE KCachegrind.
- Para monitorizar en cada punto de la ejecución el uso y procedencia de la memoria del proceso. Se ha utilizado para validar las modificaciones que afectan a la memoria estática.

Understand. Esta herramienta de análisis de código ha servido para extraer información de llamadas, dependencias entre variables y funciones, y entender el flujo de ejecución, en las etapas tempranas del proyecto (utilizado bajo licencia de estudiante).

Sonar Qube. Esta otra herramienta de sanitización y análisis informó de diversas prácticas potencialmente peligrosas y obsoletas (utilizado gratuitamente, al alojarlo en mi máquina).

Mathematica. Este sistema de cálculo simbólico se ha utilizado para la generación de parte test unitarios (se ha empleado una licencia de estudiante).

Doxygen. Este software de generación de documentación se ha utilizado para generar las páginas HTML que conforman la documentación del proyecto. Doxygen se instaló junto a Graphviz para generar los gráficos de llamadas (código abierto).

Overleaf. Este editor/compilador de LaTeX, online se ha utilizado para maquetar este documento (utilizado gratuitamente).

Jupyter. Este front-end de Python se ha utilizado para leer y generar las gráficas de tiempos de ejecución, utilizando las librerías `matplotlib` y `seaborn` (código abierto).

DIA. Se ha utilizado para generar los diagramas estructurados UML y EDP (código abierto).

Capítulo 2

1ª iteración: Puesta en marcha del propagador USM

Este capítulo se ha dividido en tres secciones. En la primera se describe el análisis del código utilizando herramientas automatizadas de análisis estático. Estas herramientas han proporcionado la información necesaria para comprender el diseño y las prácticas de programación empleadas en el propagador USM. En la segunda sección se describe brevemente el trabajo de transformación del código de las dependencias de la librería DLL, así como las acciones llevabas a cabo para su compilación y ejecución. Para finalizar, se proporcionará una lista con las acciones de mejora que se implementarán en la siguiente iteración.

2.1. Análisis del código

La primera tarea que se abordó al inicio de la primera iteración fue evaluar el código que proporciono el Dr. Cefola. Para ello se utilizaron dos herramientas de análisis estático de código: Understand y Sonar Qube. Estas herramientas ayudaron a detectar código duplicado, tamaño de archivos de código, tamaño de los métodos, calidad del software, prácticas inadecuadas,...

El código inicial constaba de

- 54.959 líneas de código.
- 2905 comentarios, la mayor parte de ellos se encuentran en ruso. Los comentarios que se encuentrn en inglés fueron añadidos por el Dr. Cefola durante la comparación entre los propagadores DSST y USM.
- 46 funciones, aunque el código de varias de ellas está comentado.
- 37 macros del preprocesador.

- 6 archivos fuente más el archivo de datos de entrada: `Hoko.h`, `Hoko0.cpp`, `Hoko1.cpp`, `Hoko2.cpp`, `Hoko3.cpp`, `Mnklib.cpp`.

Es importante destacar que al menos faltaba un fichero de constantes físicas relacionadas con el cálculo de la densidad atmosférica. Este cálculo es fundamental para evaluar el frenaje atmosférico.

El paradigma utilizado en el diseño del propagador USM fue el procedural, aunque se utilizase un compilador de C++. En la Figura 2.1 se muestra el diseño del propagador USM. Este diagrama se obtuvo utilizando Graphviz y Doxygen, aunque el código esté escasamente documentado y algunos de los comentarios estén en ruso, Doxygen puede extraer la información necesaria para generar el árbol de llamadas.

Las funciones más importantes del propagador están distribuidas en los ficheros de la siguiente manera:

- Hoko.h: Contiene macros, declaración de estructuras de E/S, y todas las cabeceras de funciones del proyecto.
- Hoko0.cpp: Contiene las funciones `KOOPLS, UV, REZ, LS, APPROK` y `SHADOW`.
- Hoko1.cpp: Contiene las funciones `sign, KEPLER, ALFDEL, GHCK, LBK, ST, C202, BIN, FINC, HANSEN, XNEWC1, QPRT, ATM, IO, I1, KATM` y `F107KP`.
- Hoko2.cpp: Contiene las funciones `FINC2` y `SKP`.
- Hoko3.cpp: Contiene las funciones `PROGNOZ` y `FORCE`.
- Mnklib.cpp: Contiene funciones matemáticas.

Los analizadores detectaron dos registros utilizados en el sistema de Entrada/Salida. Estos registros almacenan los datos y parámetros físicos iniciales, y en el caso de la de salida, almacena los datos de cada paso de integración, que representan la salida del propagador. Su estructura puede verse en la Figura 2.2.

Los analizadores de código también detectaron algunas prácticas de programación obsoletas, es de suponer que influidas por las limitaciones tecnológicas del momento en que se desarrolló. Los ordenadores disponían de poca capacidad de proceso y memoria. A continuación, se detallan estas prácticas:

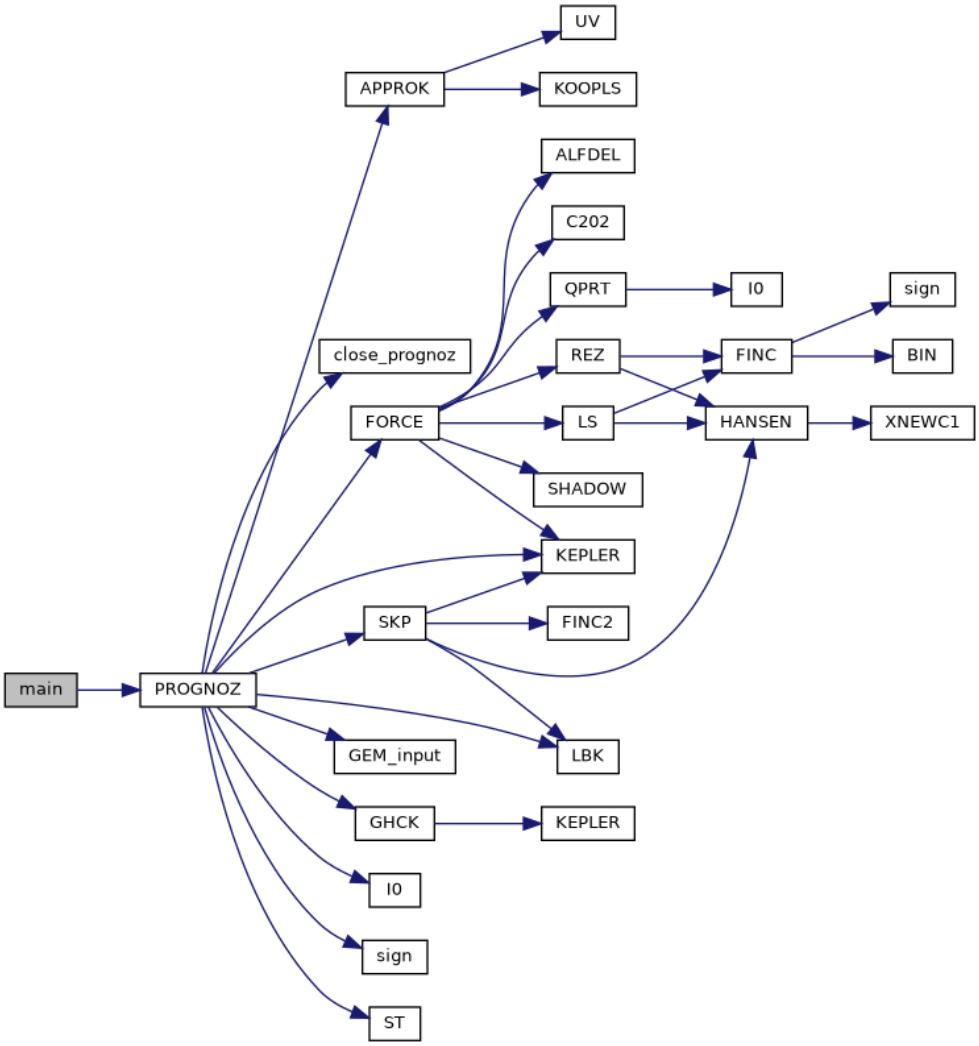

Figura 2.1: Árbol de llamadas.

Diseño confuso basado en el uso de variables globales. El uso de variables globales es una práctica habitual en el diseño e implementación de cualquier tipo de propagador orbital. En el caso del propagador USM, las variables globales han sido declaradas e inicializadas en varios ficheros. Como se puede observar, la Figura 2.3 muestra una relación jerárquica (declaración y uso) entre los diferentes ficheros que hace especialmente difícil razonar sobre las relaciones entre variables globales y funciones que las usan.

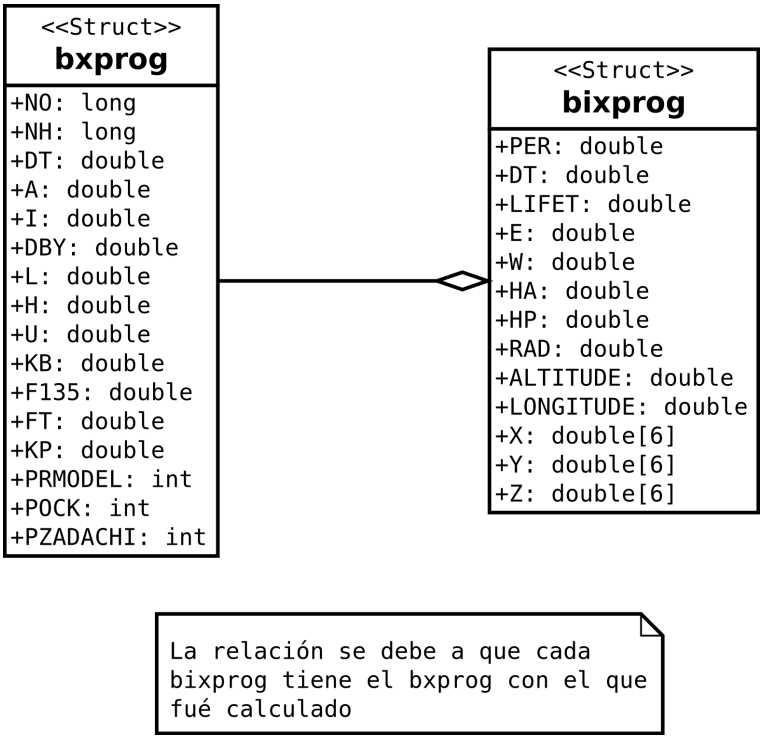

Figura 2.2: Estructuras utilizadas en el sistema de E/S.

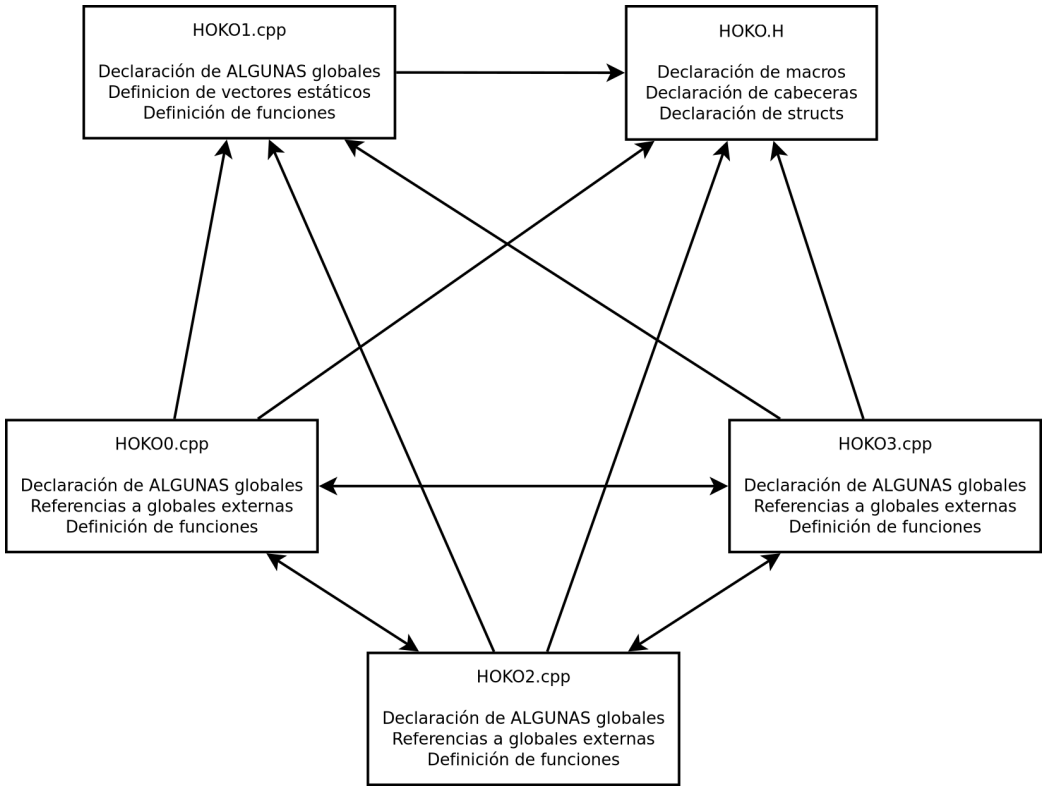

Figura 2.3: Dependencias entre los archivos originales.

Macros de preprocesador La selección del modelo de perturbaciones se realiza mediante macros del preprocesador. Esto obliga a compilar el código cada vez que se modifica el modelo de perturbaciones.

Uso de archivos binarios como memoria auxiliar. El propagador, durante su ejecución, genera varios ficheros binarios para almacenar resultados intermedios. Este modo de proceder, se justifica porque el USM fue diseñado para ser utilizado en ordenadores con recursos limitados. Además, es importante destacar, que este sistema de almacenamiento intermedio basado en ficheros binarios complicaría seriamente la integración del propagador con DACE.

Código ligado a tecnologías obsoletas. El código formaba parte de una librería DLL de Windows y se había implementado utilizando una de las primeras versiones Borland C++. Además, utilizaba algunas librerías no incluidas en el estándar de C++, como son `conio.h` o `dos.h`, o la siguiente porción de código de la función `F107KP`:

```
1         if(i_bd_fkp==-1)
2             return(i_bd_fkp);
3         if(i_bd_fkp<-1)
4        {
5         if((i_bd_fkp=d4use("f107kp"))<0)
6          {i_bd_fkp=-1;
7           return(i_bd_fkp);}
8          if((i4index("jd1957","JD1957",0,0))<0)
9          {i_bd_fkp=-1;
10          return(i_bd_fkp);}
11       }
12       db=d4select(i_bd_fkp);
```

Para finalizar esta sección, quiero destacar que durante toda la fase del análisis de código he utilizado tanto la documentación técnica como la ayuda de los tutores para comprender e incrementar mi conocimientos técnicos y teórico sobre este problema.

2.2. Compilación y puesta en marcha

Una vez finalizado el análisis del código se procedió a eliminar el código relacionado con la creación de la librería DLL. Esta tarea no presento ninguna dificultad y procedimos a compilar el código resultante. Para ello se utilizo el compilador de C++ de GNU.

La primera vez que se compilo nuestro código se produjeron dos errores y más de 50 avisos (warnings). Los errores estaban relacionados con dos funciones cuyo

código no se nos había proporcionado, aunque si aparecían sus prototipos en la documentación disponible. Sin embargo, los avisos que detectó el compilador estaban relacionados con variables no inicializadas y con dudosas practicas de programación. Por ejemplo, en algunos esquemas condicionales se utilizan condiciones como la siguiente:

```
if(XAXA=FORCE(T+CH2,Q1,PQ3))
```

donde la variable `XAXA` es de tipo entero.

Por otro lado, las funciones cuyo código no estaba disponible eran:

- get_elset_input(): Esta función lee el fichero `input.txt` que almacena las condiciones iniciales de la propagación.
- arg_lat(): Esta función calcula el argumento de latitud.

La implementación de estas dos funciones no presentó ningún problema conceptual o de diseño. La primera porque era una función de lectura de un fichero externo cuyo formato conocíamos y la segunda porque implica una relación básica en mecánica orbital, la ecuación de Kepler. El código correspondiente a estas dos funciones se muestra a continuación.

```
1  int get_elset_input(
2                       char * filename,
3                       double * epoch,
4                       int * id,
5                       int * revNumber,
6                       double * stepSize,
7                       int * stepCount,
8                       double * a,
9                       double * i,
10                      double * ascNode,
11                      double * e,
12                      double * argPerigee,
13                      double * meanAnomaly,
14                      double * ballisticCoeff,
15                      int * outputType
16                    )
17 {
18   FILE *fp;
19   char name[100];
20
21   fp = fopen(filename,"r");
22
23   if (fp == NULL) {
```

```
24      printf("Error: the file %s cannot open\n", filename);
25      printf("Abort execution\n");
26
27      return 1;
28    }
29    fscanf(fp,"%s%lf", name, epoch);
30    fscanf(fp,"%s%d",  name, id);
31    fscanf(fp,"%s%d",  name, revNumber);
32    fscanf(fp,"%s%lf", name, stepSize);
33    fscanf(fp,"%s%d",  name, stepCount);
34    fscanf(fp,"%s%lf", name, a);
35    fscanf(fp,"%s%lf", name, i);
36    fscanf(fp,"%s%lf", name, ascNode);
37    fscanf(fp,"%s%lf", name, e);
38    fscanf(fp,"%s%lf", name, argPerigee);
39    fscanf(fp,"%s%lf", name, meanAnomaly);
40    fscanf(fp,"%s%lf", name, ballisticCoeff);
41    fscanf(fp,"%s%d",  name, outputType);
42
43    fclose(fp);
44
45    return 0;
46  }
47
48
49  double arg_lat(double eccentricity, double meanAnomaly,
        double argPerigee)
50  {
51    double eccentric_anomaly;
52
53    eccentric_anomaly = KEPLER(meanAnomaly, eccentricity);
54
55    return(argPerigee + 2*atan(sqrt((1+eccentricity)/(1-
       eccentricity))*tan(eccentric_anomaly/2)));
56  }
```

Una vez implementadas las funciones y analizados los avisos se pudo compilar y ejecutar el código sin ningún tipo de errores. El propagador se comunica con el usuario mediante un archivo de entrada `input.txt`, que contiene los parámetros iniciales de cálculo. La salida del propagador se muestra en una terminal. En este momento se planteó de la corrección de los resultados obtenidos.

2.3. Validación

En este punto, acordamos que los tutores se encargarían de validar esta primera versión del propagador USM, ya que tienen una amplia experiencia en validar y verificar software de este tipo. Para ello, entre otros test procedieron a reproducir los ejemplos que se utilizaron en la referencia [2]. Estos incluían la propagación de diferentes tipos de órbitas (LEO, MEO, GEO y HEO) durante 2 años. Las comparaciones se realizaron utilizando elementos orbitales. En las Figura 2.4 se muestran el semieje y la inclinación del satélite Sirio. Éste es un satélite Geoestacionario cuyo periodo orbital es de 24 h.

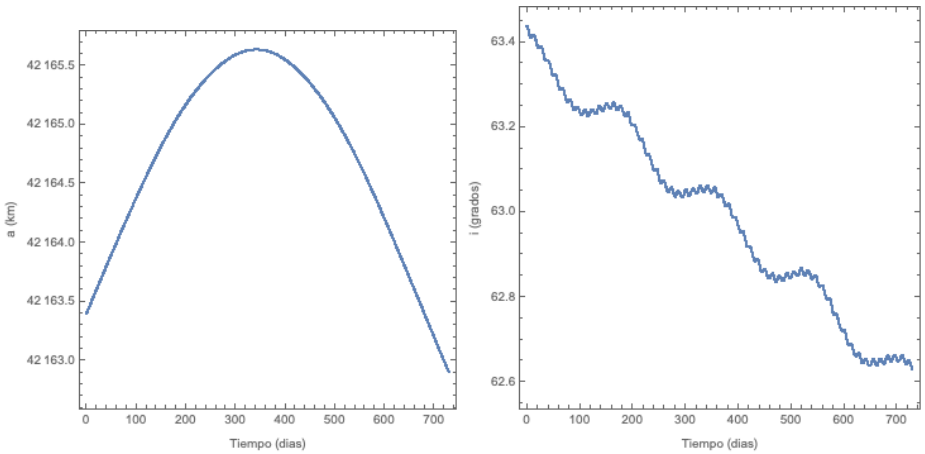

Figura 2.4: Propagación del semieje mayor y la inclinación.

Después de ajustar algunos parámetros físicos y de una exhaustiva labor de testeo, consideraron que el código resultante estaba validado y podía ser tomado como código de referencia para las siguientes actuaciones que se realizarán en este TFG. Los casos utilizados durante este proceso serán los utilizados en la siguiente iteración como test de integración.

2.4. Descomposición de tareas para la 2ª iteración

Para finalizar este capítulo, se presenta a través de un diagrama EDP (Figura 2.5) la descomposición de tareas que se va a llevar a cabo durante el proceso de refactorización de nuestro código.

Como se puede observar, entre las acciones que se incluyen en la etapa de refactorización ninguna afecta seriamente al diseño del propagador ya que éste es completamente compatible con su integración con DACE. También, se ha decidido mantener el sistema de Entrada/Salida original porque en este proyecto, de

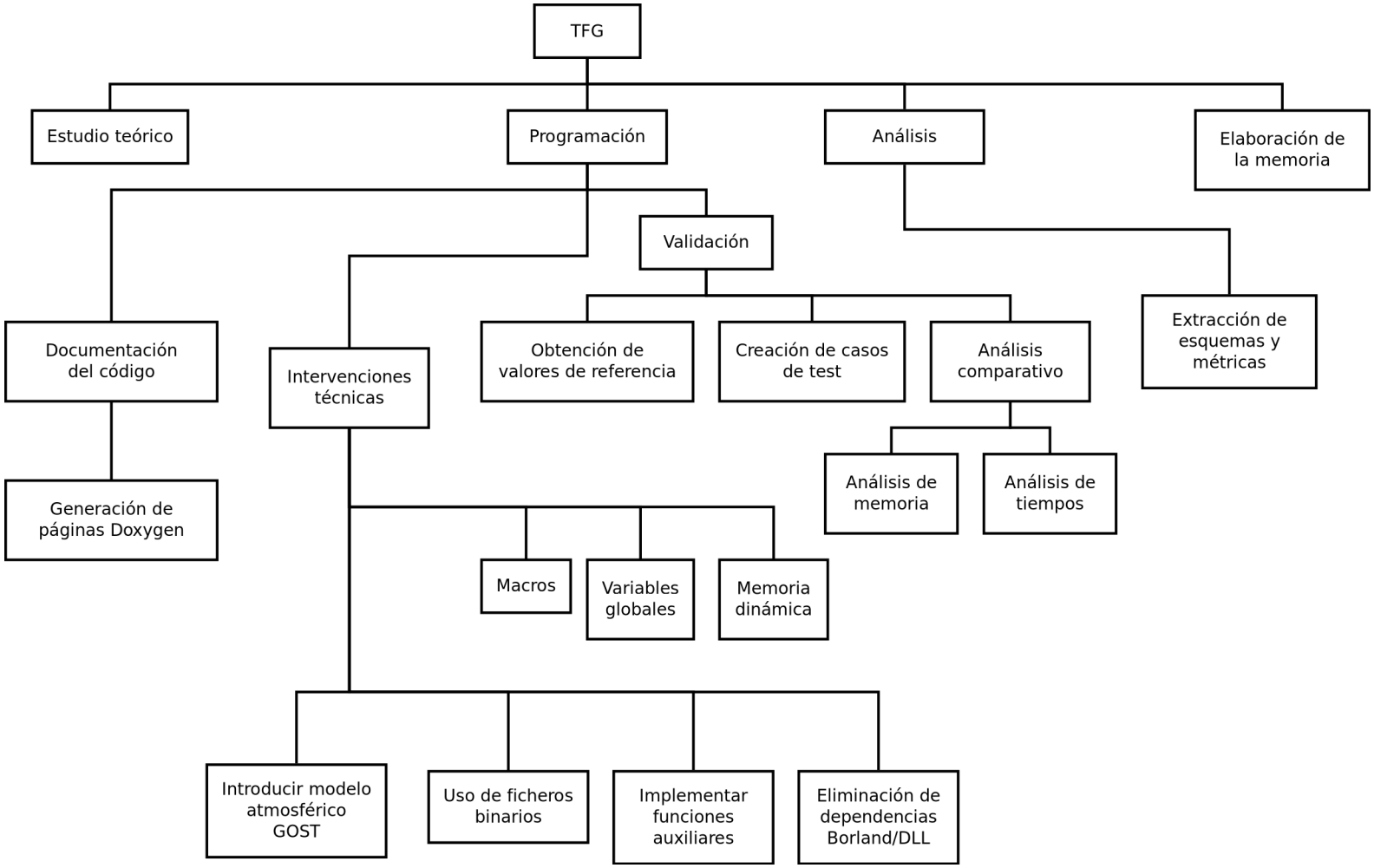

Figura 2.5: Descomposición de tareas. Segunda Iteración.

momento, no se ha planificado integrar el propagador dentro de ningún entorno operacional y, por tanto no es necesario integrarlo dentro de un entorno gráfico.

Capítulo 3

2ª iteración: Refactorización del propagador USM

En la primera sección de este capítulo se abordan los problemas detectados en la primera iteración. Inicialmente, se ha documentado el código, se han implementado los test unitarios y se ha comparado la formulación matemática con su implementación. Esta comparación nos ha servido para modificar los nombres de algunas variables y relacionarlo con su significado físico y, de este modo, construir un código más claro y legible. Durante toda esta iteración se han aplicado técnicas de verificación continua, es decir, la realización de una modificación implica que se ejecuten los test unitarios y de integración, de modo que se garantice que las funcionalidades del código generado y validado en la primera iteración no se han visto alteradas. Para finalizar, se ha analizado la influencia de la refactorización sobre la eficiencia del nuevo código.

3.1. Refactorización

Aunque en la sección 1.2 se habían planificado cuatro tareas para la segunda iteración de este TFG, la resolución de los problemas detectados y la validación numérica, se realizaron en paralelo.

3.1.1. Documentación

Generar una documentación on-line (en formato HTML) utilizando Doxygen y Graphviz fue la primera tarea que se realizó al comienzo de esta iteración. Esta documentación nos ha proporcionado un sistema ágil de acceso a todos los elementos del código y, en consecuencia, nos ha facilitado el resto de las tareas de la refactorización. Es importante destacar que Doxygen, con un mínimo esfuerzo, permite generar la documentación en otros formatos como son LaTeX, HTML, RTF, PDF, ManPages...

Durante este proceso se reutilizaron los comentarios en inglés incluidos en el código original. Estos comentarios sólo proporcionan información sobre las principales funciones del propagador, sus parámetros y sus dependencias externas.

Los problemas que nos encontramos durante el desarrollo de esta tarea estuvieron relacionados con el uso avanzado de Doxygen y con algunos de los elementos del código que no se encontraban documentados en el código original.

A continuación se muestra un ejemplo de como se ha documentado el código de este proyecto:

```
/** \brief Calculation of function b = b * (a!/(k!*(a-k)!
    ===> b * Binomial Coefficient(a,k))
 *
 * \param INPUT      double B
 * \param INPUT      int A
 * \param INPUT      int K
 * \return The result of the operation
 *
 */
double BIN(double B, int A, int K)
{
  double S;
  int I, A1;

  S = ((A < K) ? 0 : B);
  I = A - K;
  if(K > I)
    K = I;
  A1 = A + 1;
  for(I = 1; I <= K; I++)
    S *= (A1 - I) / (double) I;
  return (S);
}
```

Los macros `\brief, \param` y `\return` permiten especificar la función, describir los parámetros e indicar el valor que devuelve.

3.1.2. Actuación general

Como ya se ha mencionado, los problemas que se detectaron en la primera iteración son de diversa naturaleza y se encuentran distribuidos a lo largo de todo el código. Para facilitar la organización, el control de cambios y la verificación continua del código, se decidió dividir el proceso de refactorización en pequeñas tareas,

donde en cada tarea se trata, a lo sumo, a una función. De esta forma, se podrían mantener acotados los errores que se pudieran cometer en cada transformación del código y se garantizaría que cada cambio no afectara al comportamiento de la función.

Debido a mi escasa experiencia en este tipo de problemas, los tutores aconsejaron utilizar una estrategia *de afuera hacia adentro* (Figura 2.1), es decir, se comienza por las funciones que no tienen ninguna dependencia y se avanza hacia las funciones que se encuentren en un nivel superior, de modo que el proceso termina cuando se alcanza la función principal.

Para cada función se crearon los casos de test utilizando el código generado en la primera iteración, o bien se implementaron los métodos teóricos en el sistema de cálculo Mathematica. A continuación, se procedió a formatear el estilo del código utilizando las herramientas proporcionadas por el IDE JetBrains CLion. Por último, se reemplazaron los macros del preprocesador que seleccionaban el modelo de perturbaciones en tiempo de compilación por esquemas condicionales.

A continuación se muestra un ejemplo de test unitario:

```
/** \brief XNEWC1 function test. Reference values taken from
    execution of the original code.
 *
 */
void newcombTest()
{
  double *value = (double *)calloc(1,sizeof(double));
  double *derivative = (double *)calloc(1,sizeof(double));
  double diffVal = 0.0;
  double diffDer = 0.0;

  double parameters[][4] =
      {
          {2,2,0,0.01},
          {2,0,0,0.01},
          {2,-2,0,0.01},
          {3,3,0,0.01},
          {3,1,0,0.01},
          {3,-1,0,0.01},
          {3,-3,0,0.01},
          {4,4,0,0.01},
          {4,2,0,0.01},
          {4,0,0,0.01},
          {4,-2,0,0.01},
          {4,-4,0,0.01},
```

```
25      };
26
27  double results[][2] =
28      {
29          {0.00025000000000000001,0.047499999999999959},
30          {1.0001500000000001,0.028499999999986869},
31          {0.00025000000000000001,0.047499999999999959},
32          {-4.3750000000000005e-06,-0.001185624999999999},
33          {-0.025001875,-2.5005081249999983},
34          {-0.025001875,-2.5005081249999983},
35          {-4.3750000000000005e-06,-0.001185624999999999},
36          {7.875e-08,2.7082124999999981e-05},
37          {0.00052502625000000003,0.099759027375000045},
38          {1.0005000187499999,0.095006448124879128},
39          {0.00052502625000000003,0.099759027375000045},
40          {7.875e-08,2.7082124999999981e-05},
41      };
42
43  printf("Executing XNEWC1 test, displaying tolerances for
     each case: \r\n");
44  for(int i = 0; i < 12; i++)
45  {
46    (*value) = XNEWC1(parameters[i][0],parameters[i][1],
     parameters[i][2],parameters[i][3]);
47    (*derivative) = (*value-XNEWC1(parameters[i][0],
     parameters[i][1],parameters[i][2],parameters[i][3]-0.001))
     /0.001;
48    diffVal = abs(*value - results[i][0]);
49    printf("Input: %f %f %f %f   Difference (value): %e\r\n",
      parameters[i][0], parameters[i][1], parameters[i][2],
     parameters[i][3], diffVal);
50    assert(diffVal < _TOL_);
51    diffDer = abs(*derivative - results[i][1]);
52    printf("Input: %f %f %f %f   Difference (Derivative): %e\
     r\n", parameters[i][0], parameters[i][1], parameters[i
     ][2], parameters[i][3], diffDer);
53    assert(diffDer < _TOL_);
54  }
55
56  free(value);
57  free(derivative);
58 }
```

Es importante destacar que ninguna de las actuaciones que se han propuesto en este TFG afectan a la precisión del propagador, por lo tanto sólo se considerara que los test de integración han sido superados si se alcanza la precisión de la máquina.

3.1.3. Solucionar los problemas detectados

Una vez finalizadas las actuaciones generales, se procedió a resolver los problemas detectados en la fase del análisis del código (Sección 2.1).

Sistema de variables globales

Las actuaciones generales que se aplicaron no tenían por objeto modificar el sistema de variables globales original. La declaración de este tipo de variables todavía permanecía al comienzo de los ficheros `*.cpp` y seguían obscureciendo el código, como se puede observar en la figura 2.3.

Para resolver este problema se agruparon todas las definiciones de las variables globales en un único fichero de cabecera llamado `globals.h`. De este modo, las funciones que necesiten alguna de estas variables las declararan en la zona de declaración de variables, anteponiendo al tipo el modificador `extern`. Esta reorganización hace que la relación entre funciones y variables globales quede explícita.

La identificación inicial de estas dependencias fue sencilla. Después de trasladar todas las declaraciones de las variables globales a `globals.h`, el inspector de código del IDE JetBrains CLion detectó, en timepo real y sin necesidad de compilar el código, que dentro de las funciones algunas variables no estaban declaradas. Para resolver este problema sólo fue necesario declarar las variables dentro del cuerpo de la función y utilizar el modificador `extern`.

Sin embargo, esta transformación aparentemente sencilla, supuso varios días de retraso. El problema fue provocado por no incluir correctamente el fichero de cabecera dentro del proyecto. El compilador durante el proceso de enlazado informaba de variables que habían sido declaradas múltiples veces. Es decir, el compilador detectó que el fichero de cabecera estaba incluido en todos los ficheros `*.cpp`. Este error se soluciono cuando `globals.h` se eliminó de todos los ficheros de código excepto del que contiene la función `main`. Sólo la poca experiencia en el uso de variables globales y su organización dentro de un proyecto en C/C++ puede justificar este problema.

Eliminación de ficheros binarios

El propagador original utiliza un sistema basado en ficheros binarios para almacenar resultados intermedios, lo cual supone un serio obstáculo para su futura integración con la librería DACE. Las funciones implicadas el manejo en este en sistema son `PROGNOZ`, `FORCE` y `SKP`, que además son las más complejas del propagador.

Tras analizar en profundidad este sistema de almacenamiento intermedio, se contemplaron varias alternativas para sustituir el uso de este sistema, intentando modificar lo mínimo posible la lógica interna y sin utilizar memoria dinámica. Las conclusión a las que se llego después de este análisis es que el sistema simula al funcionamiento de una base de datos. Durante este fase, y con el fin de entender mejor las transferencias que se realizaban con los ficheros, se construyó un sistema paralelo basado en memoria estática como una posible alternativa.

La ejecución de los test de integración confirmó que la solución basada en memoria estática reproduce el comportamiento de los sistemas binarios, queda como tarea pendiente que es la solución óptima.

Reestructuración de macros

Otra acción que se llevo a cabo con el fin de clarificar el código, fue la agrupación de todos los macros del preprocesador en un fichero de cabecera llamado `directives.h`. Esta tarea permitió unificar y dar coherencia a los valores de las constantes físicas. Por ejemplo, en la versión original del propagador se definen las constantes π y 2π utilizando macros. La variable π utilizaba el valor de esta constante proporcionado por la librería matemática `math.h`, `#define PI M_PI`, sin embargo para el valor de 2π utilizaba un valor numérico `#define PI2 6.28318530719`.

Reducción de uso de memoria dinámica

La versión original del propagador fue diseñada para operar en ordenadores con pocos recursos, de modo que el sistema de gestión dinámica de memoria se convirtió en un mecanismo útil para incrementarlos. En la implementación del propagador USM, la memoria dinámica se utilizó en la declaración de varios vectores de tipo `double`, de tamaño relativamente reducido, y que con los recursos de los sistemas actuales se pueden declarar como vectores estáticos. Estos vectores aparecen en las funciones que implementan los efectos producidos por el campo gravitatorio terrestre y por la influencia del Sol y de la Luna sobre un RSO.

La trasformación de los vectores dinámicos a estáticos fue casi inmediata y no supuso ningún problema.

Traducción de campos y parámetros

Uno de los problemas recurrentes con los que nos hemos encontrado a lo largo del desarrollo de este TFG es la poca claridad del código. Por ejemplo, los identificadores utilizados en el sistema de Entrada/Salida (Figura 2.2).

Es por tanto, que nos propusimos mejorar los identificadores que están implicados en este sistema. Para ello se contó con la ayuda del documento [1], que contiene las descripciones en inglés de dichos campos y estructuras. El resultado final puede verse en la Figura 3.1.

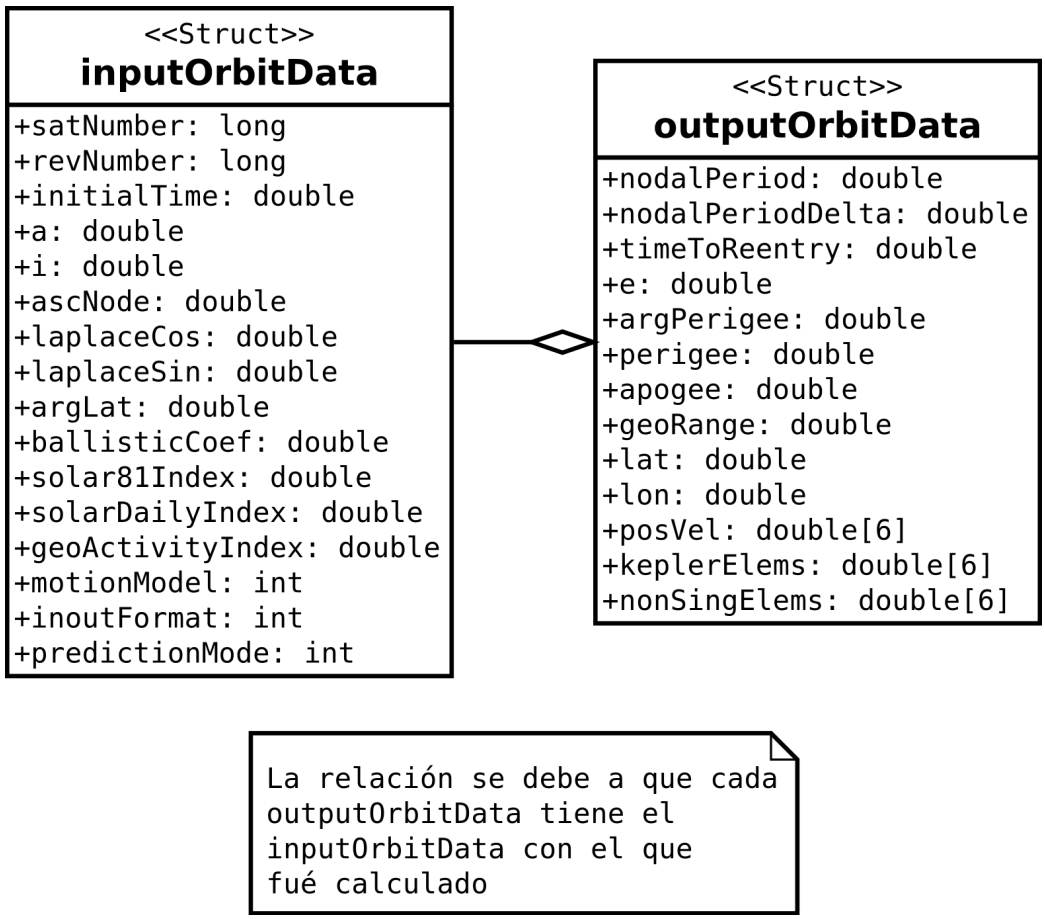

Figura 3.1: Estructuras de Entrada/Salida refactorizadas.

Desde el punto de vista técnico no surgió ningún problema significativo, pero costó más tiempo de lo previsto debido al desconocimiento de los conceptos básicos implicados en el sistema de Entrada/Salida.

3.2. Análisis de eficiencia

En esta sección se analiza la influencia de las actuaciones realizadas sobre la eficiencia temporal y el uso de memoria del código, comparando las versiones generadas durante el desarrollo de este TFG.

3.2.1. Análisis de velocidad de ejecución

Para realizar las mediciones precisas del tiempo de ejecución del código se ha utilizado la clase `high_resolution_clock` que pertenece a la librería estándar `<chrono>`. El procesador sobre el que se ha medido la eficiencia es un AMD Ryzen 5 2600 con 16GB de RAM, ejecutado sobre un S.O. Manjaro Linux 64-bit y compilado con `g++ 10.1`.

Se consideraron los códigos obtenidos al final de la primera y de la segunda iteración. Las dos versiones del propagador se han compilado activando el sistema de optimización del compilador, opción `-O3`, y sin ella. Cada una de estas cuatro versiones se ha ejecutado 1000 veces de forma automática, y se han medido sus tiempos de ejecución. La órbita propagada es de tipo LEO (Low Earth Orbit) que ha sido integrada durante un periodo de 2 días y en la que se han considerado todas las perturbaciones disponibles. La Figura 3.2 muestra el histograma de los tiempos de ejecución de este experimento, mientras que la Tabla 3.1 los principales estadísticos. Todos los tiempos son en milisegundos.

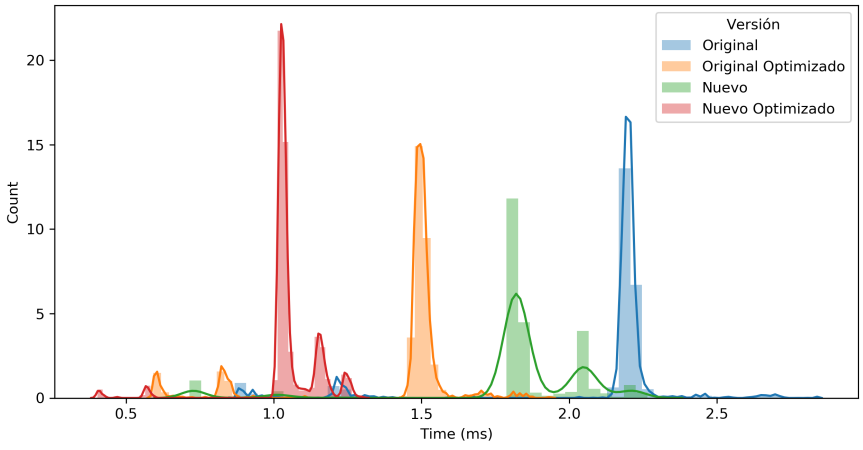

Figura 3.2: Histograma de los tiempos de ejecución de las cuatro versiones del código.

Como se muestra la Tabla 3.1, la refactorización del propagador USM ha mejorado su eficiencia en ambas versiones.

	Original	Original Opt.	Nuevo	Nuevo Opt.
Media	2.085882	1.409197	1.820763	1.047698
Desviación estándar	0.368630	0.270591	0.297183	0.116735
Mínimo	0.867391	0.591977	0.714452	0.403126
Máximo	2.837403	1.933162	2.265283	1.299971

Tabla 3.1: Principales estadísticos del análisis de la eficiencia temporal.

3.2.2. Análisis de uso de memoria

Se ha utilizado Massif, una herramienta integrada en Valgrind, para analizar como se ha visto afectada la gestión de memoria por la refactorización. En las Figuras 3.3 y 3.4, se puede observar que tras las intervenciones técnicas realizadas, la memoria usada desciende desde 162KiB del original hasta 75KiB de la nueva implementación.

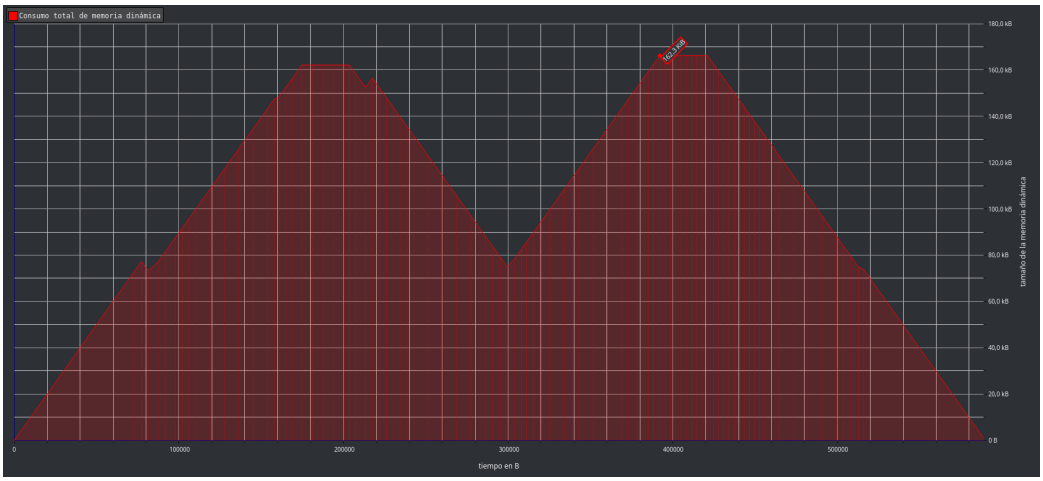

Figura 3.3: Uso de memoria del original.

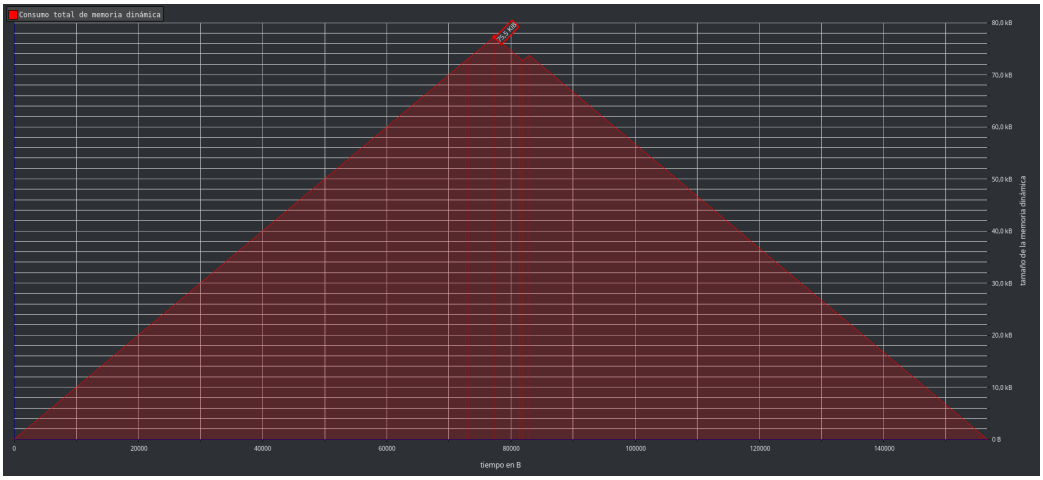

Figura 3.4: Uso de memoria de la nueva versión.

3.3. Resultado de la refactorización

Para finalizar, en esta sección se va a describir el estado en el que se encuentra el propagador USM después de la refactorización realizada a lo largo del desarrollo de este TFG. Todas las actuaciones realizadas están encaminadas a facilitar el trabajo de futuros desarrolladores en la integración, extensión o mejora de este propagador.

Portabilidad y estandarización Las intervenciones realizadas en la primera iteración han transformado el sistema inicial en un programa portable, independiente y compatible con los principales compiladores y sistemas operativos modernos. Estas intervenciones posibilitan la ejecución del código en sistemas compatibles con las herramientas con las que se quiere integrar en sucesivas iteraciones.

Documentación La documentación que acompaña a este TFG en formato HTML permite la navegación y búsqueda de todos los elementos internos del propagador. Además, los bloques internos de documentación Doxygen permitirán generar documentación automáticamente en la variedad de formatos que soporta Doxygen. Los tutores han considerado muy importante la existencia de una forma universal, y lo más completa posible de documentar tanto la manera de comunicarse con el propagador cara a usuarios futuros, como las funciones internas como base al desarrollo futuro.

Reorganización estructural Aunque el diseño interno no se ha modificado, las actuaciones han mejorado sustancialmente la claridad del código respecto a la versión inicial. La reestructuración de macros y variables globales ha sentado las

bases para futuras intervenciones en materia de diseño interno y la extensión de la parametrización disponible en tiempo de ejecución, respectivamente.

A modo ilustrativo, las dependencias entre los ficheros y la separación de intereses, en comparación con la estructura original (Figura 2.3), se muestran en la Figura 3.5.

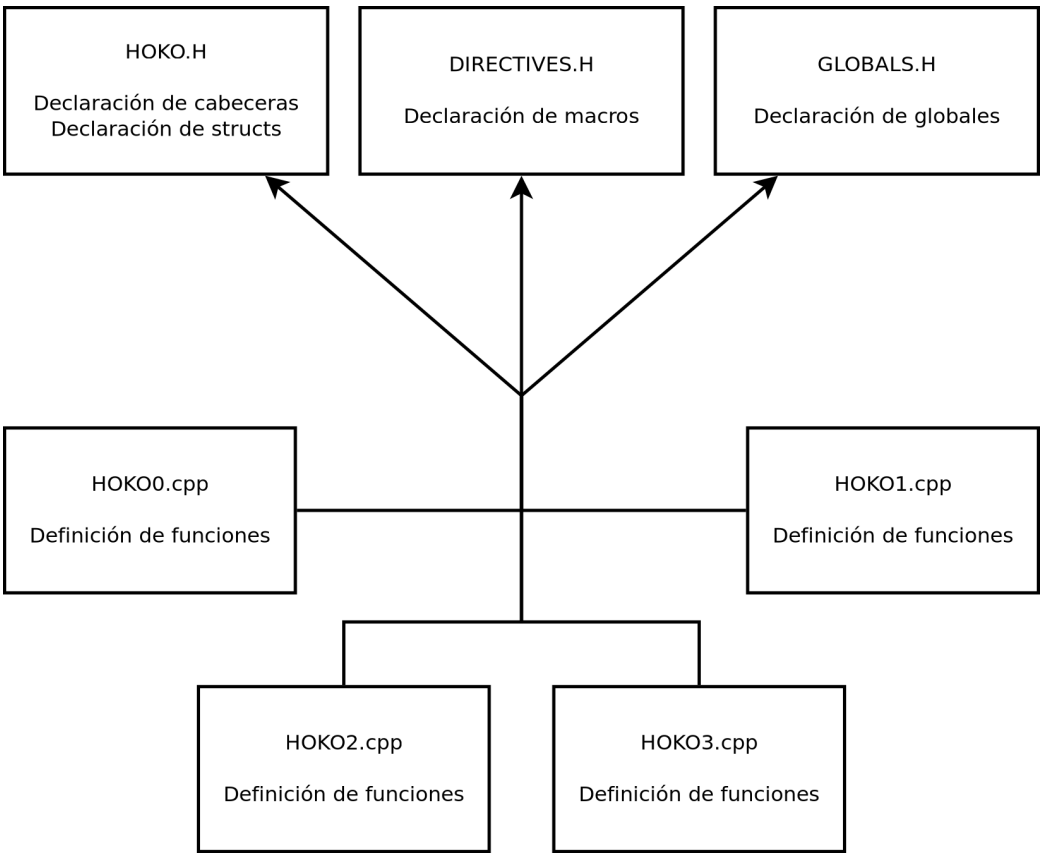

Figura 3.5: Organización del código final

Traducción de campos y parámetros Los nuevos nombres de los campos de las estructuras de E/S, así como de las variables del punto de entrada (`int main()`), y los parámetros formales de las funciones (Siempre que ha sido posible) han mejorado la comprensión del código, y han dotado de un significado físico más claro al funcionamiento interno. A modo comparativo, la Figura 3.1 muestra las nuevas estructuras de E/S.

Memoria optimizada La nueva versión del propagador hace uso de las capacidades de los computadores modernos, si necesidad de recurrir a memoria secundaria, en forma de archivos binarios. Además, se ha disminuido el uso de memoria

dinámica, exceptuando los casos donde es estrictamente necesario. Estas actuaciones han sido las principales reductoras del tiempo de ejecución, junto con las optimizaciones a nivel de compilador.

Validación numérica La nueva versión del propagador incluye una serie de test unitarios, que se suman a los test de integración, comparables con las versiones anteriores, ampliamente utilizadas y validadas durante años. Se facilita así los futuros trabajos de mejora de los algoritmos internos, al contar con un conjunto de resultados de referencia.

Jerarquía de funciones Debido a la inclusión de los test unitarios y las funciones auxiliares, el nuevo árbol de funciones es ligeramente mayor que el original, como se puede observar en la Figura 3.6. El resto de la jerarquía original no se ha modificado, a excepción de las funciones excluidas, como se explica en el capítulo de Conclusiones. En la Figura 3.6 se muestra el nuevo árbol de llamadas. El esquema ha sido generado incluyendo el sistema de test unitarios.

Sin embargo, por falta de tiempo las siguientes tareas planificadas al inicio del TFG no han podido ser completadas:

Modelo GOST La exclusión del modelo GOST para modelar el frenaje atmosférico, y con ello, las funciones `F107KP`, `ATM` y `KATM` se ha debido a la falta de un archivo externo requerido por estas funciones, y que no ha podido obtenerse dentro del plazo.

Selección de perturbaciones en tiempo de ejecución En la versión final las perturbaciones se siguen configurando mediante macros. Se hace de tal modo para mantener la compatibilidad con la versión de la primera iteración.

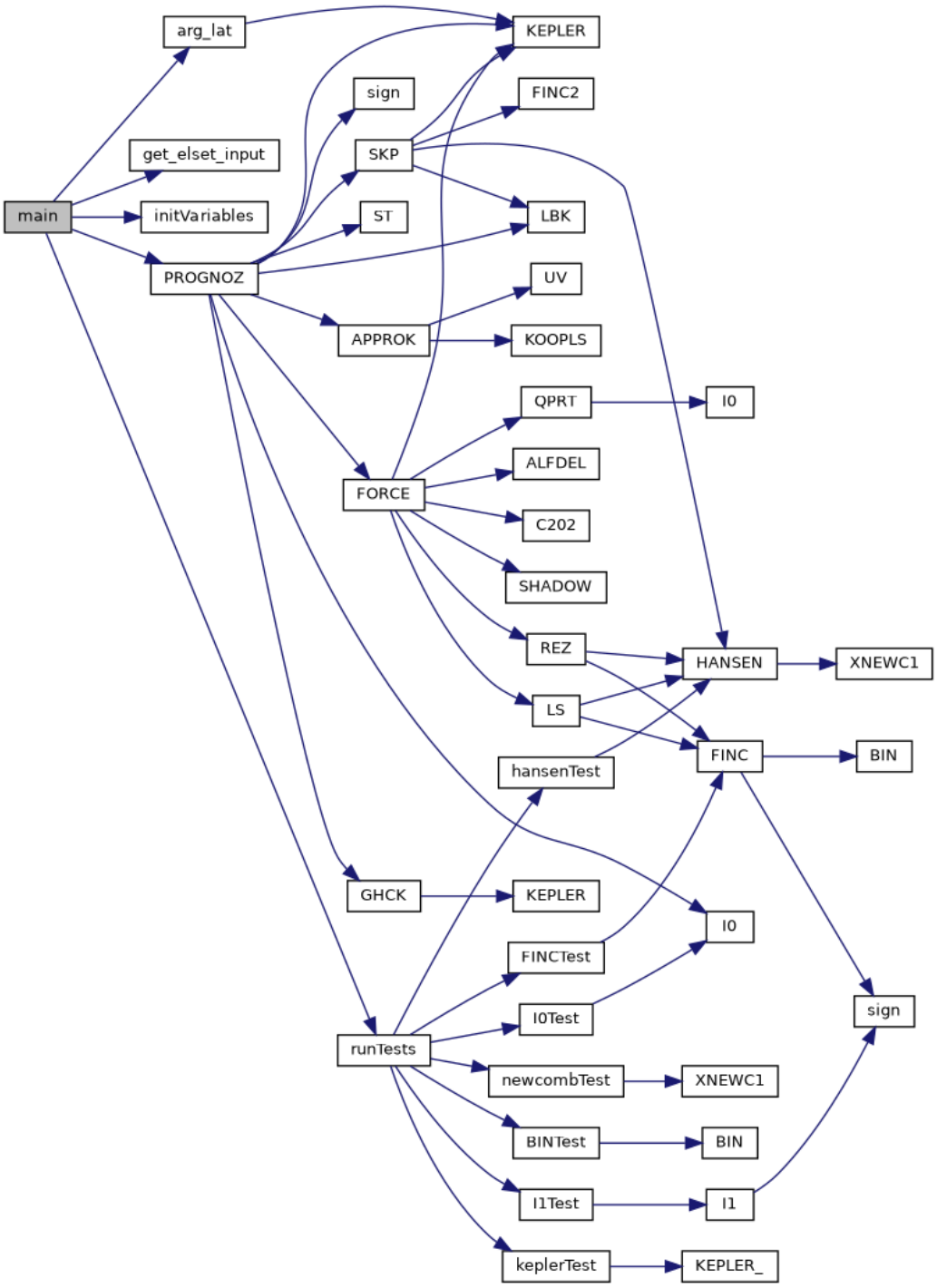

Figura 3.6: Árbol de funciones del propagador refactorizado.

Capítulo 4

Conclusiones

Este TFG es parte de un proyecto más ambicioso que pretende actualizar el propagador semi-analítico USM, Universal Semi-Analytical Model, y adaptarlo a las necesidades actuales del sector espacial. En este TFG se abordan las dos primeras iteraciones del proyecto.

La primera, su puesta en marcha, se realizó con éxito. Se efectuaron las acciones necesarias para que el propagador fuese otra vez operativo, salvo las funciones que implementan el efecto del frenaje atmosférico. Esta parte del trabajo no se ha podido realizar porque no se dispone en estos momentos de un fichero de constantes físicas. Estos parámetros físicos son necesarios para el cálculo de la función de densidad atmosférica. A continuación, los tutores validaron exhaustivamente el código. Esta tarea fue fundamental para garantizar el éxito de las iteraciones posteriores y generar los tests de integración necesarios.

La refactorización de este código es la segunda tarea que se trata en este TFG. Un objetivo claro dirige esta tarea, la integración del propagador USM con DACE, una librería que implementa el cálculo de incertidumbres utilizando la técnica llamada de álgebra diferencial. Reorganización del sistema de variables globales, eliminación del sistema de reserva dinámica de variables y limpieza del código fueron algunas de las tareas que se realizaron. Al mismo tiempo que se revisaba el código se han implementado test unitarios y se ha comparado la formulación matemática con el código de su implementan. También se ha construido un sistema de documentación automática basada en Doxygen. Esta documentación es en una herramienta básica para el mantenimiento y el escalado del propagador. Cabe destacar que la ejecución de los tests unitarios y de integración se realizaron después de cada actuación sobre el código. Finalmente, utilizando Valgrind se analizó el efecto de la refactorización sobre la eficiencia del nuevo código.

4.1. Conocimientos adquiridos

Considero que este proyecto ha sido muy beneficioso para mi formación como futuro ingeniero en informática, así como ha supuesto un reto a mis habilidades aprendidas durante mi formación en esta universidad, debido a la extensión y naturaleza científica del código, el análisis teórico previo requerido, la necesidad de establecer una rutina ágil de intercambio de información con los tutores, la búsqueda y solución de problemas técnicos, la necesidad de aprender a redactar una memoria técnica, la identificación y posterior reemplazo de patrones de diseño obsoletos y por ser la primera vez que me enfrento a un problema de código heredado.

Por otra parte, considero que la realización de este proyecto me ha aportado más confianza en mis habilidades para enfrentarme a problemas reales en el mundo laboral, y me ha enseñado a perder el miedo a investigar, probar, documentarme sobre un problema en concreto para aportar una solución adecuada.

AL finalizar este proyecto, he obtenido más conocimientos sobre los lenguajes de programación C y C++, la aritmética del computador, tratamiento de arquitecturas con variables globales, depuración y localización de errores, maquetado de documentos con LaTeX, documentación de código con Doxygen, y conocimientos básicos sobre la propagación orbital.

Por último, deseo recalcar que la rutina de trabajo propuesta por los tutores ha sido beneficiosa para aprender a tratar con código existente de una manera estructurada y ordenada. Ha sentado las bases de una forma de trabajar en el que puedo sentirme seguro de la corrección del trabajo anterior mediante el uso de test unitarios y de integración, además de una documentación lo más completa posible.

4.2. Desviaciones temporales

Durante el desarrollo del proyecto han surgido una serie de problemas cuya solución ha diferido de lo inicialmente estimado. Las desviaciones se deben principalmente al bajo nivel de detalle de las estimaciones iniciales. Un análisis previo, más detallado, a la primera iteración hubiera ahorrado desvíos y costes temporales.

Cabe destacar que debido a la situación de confinamiento debido al estado de alarma, y a la incertidumbre provocada por la evaluación no presencial, la productividad como programador se vio seriamente perjudicada desde el inicio del confinamiento hasta la entrega de actas del segundo semestre.

El estudio teórico de los fundamentos del propagador realmente supuso dos semanas de trabajo, una más de lo previsto, debido al bajo nivel de conocimientos matemáticos. Inicialmente se propuso un estudio con la documentación referenciada en este TFG, sin embargo, no fue posible sin ayuda adicional, que dotase de sentido a las formulaciones. Las tutorías como medio de aprendizaje de los conceptos de una manera más adecuada a mis conocimientos iniciales ha sido determinante en la comprensión de estos conceptos básicos.

Por otra parte, el análisis de código se completó en una semana, por lo que la línea temporal al finalizar la tercera semana era correcto. El uso de herramientas de análisis, junto con tutorías ayudaron a que esta fase se completase antes de tiempo.

La documentación, inicialmente planeada para una semana, pudo completarse en tres días, debido al trabajo de documentación anterior del Dr. Cefola. De igual manera, el documento [1], fue de gran utilidad al documentar las estructuras de E/S.

Sin duda, el mayor desvío temporal se debe a la resolución de los problemas detectados en la primera iteración, debido a la nula experiencia con este tipo de programas. De nuevo, la búsqueda de información en Internet y las tutorías evitaron un desvío superior. Esta fase supuso 5 semanas de trabajo, dos mas que las previstas.

El desarrollo de los test unitarios y de integración se pudo realizar en paralelo con la resolución de problemas detectados en la primera iteración.

La figura 4.1 muestra la dedicación temporal real del proyecto, debido a los problemas descritos antes:

Tareas	Inicio	Final	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10
Estudio teórico	03/02/20	16/02/20										
Análisis de código	17/02/20	23/02/20										
Extracción de la librería DLL	24/02/20	15/03/20										
Desligar el código a Borland	16/03/20	05/04/20										
Conseguir una versión funcional	30/03/20	12/04/20										
Tareas	Inicio	Final	S11	S12	S13	S14	S15	S16	S17	S18	S19	S20
Generar documentación	13/04/20	19/04/20										
Resolver problemas detectados en análisis	20/04/20	24/05/20										
Realizar test unitarios e integración	20/04/20	24/05/20										
Analizar eficiencia de la versión final	25/05/20	31/05/20										
Elaboración de la memoria	01/06/20	21/06/20										

Figura 4.1: Dedicación temporal real

Apéndice A

Elementos orbitales

Los elementos orbitales $(a, e, i, \Omega, \omega, M)$ son los datos necesarios que permiten determinar la forma, el tamaño y la orientación de la órbita de un RSO alrededor dela Tierra y que determinan su posición en cada instante. Estas cantidades se muestran en la Figura A.1. a representa el semieje mayor de la órbita, e la excentricidad, i la inclinación del plano orbital Ω el argumento del nodo, ω el argumento del perigeo y M la anomalía verdadera.

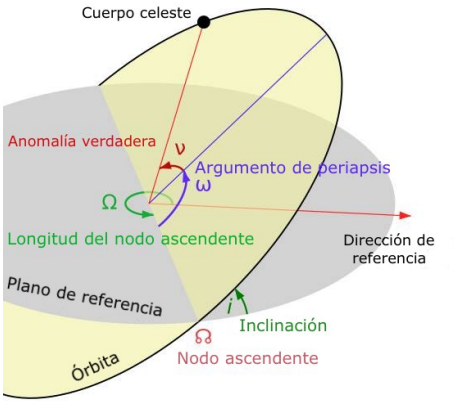

Figura A.1: Elementos orbitales [4]

En función de los elementos orbitales el argumento de latitud θ se define como $\theta = \omega + f$. f es la anomalía verdadera y se relaciona con la anomalía media M a través de la ecuación de Kepler:

$$M = E - e \sin E,$$

donde E representa la anomalía excéntrica.

Apéndice B

Reuniones

En este anexo, se muestra el diario de reuniones con los tutores: Fecha y objetivos.

3 de Febrero Primera reunión del proyecto, se introduce el problema, y el funcionamiento del propagador, se explican los problemas, antecedentes, objetivos y procedimientos del proyecto. Se propone el estudio de la documentación referenciada en este TFG, y el árbol de ejecución del propagador. Seguidamente se abordan los aspectos de organización del proyecto, como una descomposición de tareas muy básica (Ej: Documentación, memoria, pruebas, intervenciones), y una planificación temporal inicial.

10 de Febrero Debido al bajo nivel de conocimientos en Matemáticas, esta reunión se centra en explicar, adecuándose a la situación, el fundamento teórico del propagador, para asegurar un conocimiento básico que permita continuar con el proyecto, a la vez que se instruye en el problema que el propagador intenta resolver.

17 de Febrero Se propone el estudio de los nombres y parámetros formales de las funciones, para proponer nombres alternativos que faciliten el trabajo futuro. También se propone la generación de una versión sin funcionalidad, que tenga el mismo árbol de ejecución, que será el primer paso en la creación de la versión resultante de la primera iteración. Por último se propone un criterio alternativo de agrupación de las funciones.

24 de Febrero Se revisa el cumplimiento de las tareas propuestas en la reunión anterior. Seguidamente, se explica la problemática de las librerías DLL en mas detalle, se proporciona documentación necesaria para la extracción del código y se enumeran los problemas que potencialmente pueden surgir durante esta intervención, debido a la nula experiencia con este tipo de problemas. Tras esto, se

proponen las tareas concretas de la extracción del código de las librerías.

28 de Febrero Se revisa el estado de las tareas pendientes para la intervención actual. Por ahora, no suceden contratiempos en esta fase. Se continúa con las explicaciones técnicas necesarias para el trabajo hasta el final de esta fase, y se proponen las tareas concretas para finalizar la extracción del código del DLL.

16 de Marzo Se revisa el resultado de las tareas de extracción del código de las librerías DLL. Seguidamente, se introduce la problemática del grado de acoplamiento del código al compilador Borland, y el uso de librerías no estándar que dificultan la portabilidad. Se proporciona la documentación de dichas librerías, del compilador Borland, junto con la documentación de las librerías estándar C/C++ para identificar las transformaciones que garanticen la equivalencia funcional del código. A continuación se establecen unas pautas de transformación para esta intervención, junto a las tareas específicas para esta fase.

30 de Marzo Se revisa el progreso de las tareas desde la última reunión. Se proponen nuevas tareas específicas dentro de la fase de separación del código de las librerías no estándar y Borland, junto a otra serie de indicaciones y tareas con el objetivo de conseguir una versión funcional, mediante la identificación de problemas menores de compilación.

5 de Abril Se evalúan las tareas realizadas en la fase de separación de código de Borland, y se revisa el progreso de las tareas para conseguir una versión que no produzca errores de compilación, y sea funcional. Se identifican los últimos problemas de compilación, y se establecen las últimas tareas para terminar la fase.

13 de Abril Se finaliza la primera iteración del proyecto. Los tutores presentan los resultados de validación numérica con éxito, por lo tanto, se puede continuar inmediatamente con la segunda iteración sin necesidad de corregir el programa. Se concluye que todas las transformaciones de la primera iteración han resultado exitosas. Se decide realizar la documentación en Doxygen antes de continuar, y para ello se establecen las tareas de esta fase.

20 de Abril Se comprueba el correcto funcionamiento de las herramientas de documentación, y se verifica que todos los elementos anteriormente documentados por los Dres. Yurasov y Cefola se encuentran documentados en Doxygen. Esta reunión es más extensa de lo habitual, debido a que se prevé que la fase de resolución de problemas detectados en la primera iteración, junto con la validación continua de las intervenciones puede ser la más extensa de todo el proyecto. Debido a lo anterior, los tutores establecen pautas de transformación del código, captura y generación de resultados numéricos correctos y creación y ejecución de test unitarios. Siguiendo la estrategia propuesta por mis tutores, se comienza a realizar

la transformación y validación continua de las funciones puras KEPLER, HANSEN, XNEWC1, FINC, IO e I1.

28 de Abril Se verifican los resultados de los test unitarios de las funciones verificadas anteriormente, y las transformaciones de estas. Se establecen nuevas tareas de transformación del resto de funciones del sistema, exceptuando FORCE y PROGNOZ.

7 de Mayo Se verifican los cambios en las funciones propuestas en la reunión anterior. Se propone completar la transformación de las funciones restantes, con el objetivo de comprobar los resultados completos del sistema.

1 de Mayo Se completa la transformación del código, pero se detecta que los resultados globales no concuerdan con la versión de la primera iteración. Los tutores instruyen sobre estrategias posibles errores cometidos, para detectar los fallos de transformación. Se comienza con una fase de depuración intensa, para averiguar el origen de las desviaciones numéricas.

5 de Mayo Se reportan varios fallos en la inicialización de algunas variables globales, que causaban las desviaciones numéricas detectadas. Se consiguen los resultados numéricos correctos. Los tutores comienzan a establecer pautas y objetivos para la reestructuración de variables globales y macros, como objetivo final de la transformación del código.

9 de Mayo Se realiza el primer intento de reestructuración de variables globales sin éxito. Debido a el desconocimiento del funcionamiento interno de inclusión de ficheros de cabecera, el enlazador no consigue generar un binario, por lo que se comunica a los tutores, que me instruyen en el funcionamiento del enlazador.

20 de Mayo Tras unos retrasos por mi parte en la solución del problema del enlazador, y basándome en los conocimientos aportados por mis tutores, y con la ayuda de foros especializados en Internet, depuro varios fallos de declaraciones e inicializaciones realizados en el primer intento de reestructuración de variables globales.

22 de Mayo Se llega a la solución adecuada de reestructuración de variables globales, con el apoyo de los tutores. Se llega a la versión numéricamente correcta, y estructuralmente aceptable, dentro del alcance del proyecto. Se da por concluida la transformación del código.

25 de Mayo Se comienza con la fase de análisis de eficiencia. Inicialmente, se obtienen resultados incoherentes con respecto a lo esperado, ya que la metodología que seguí inicialmente no era válida. Tras comunicárselo a los tutores, ellos me informan del proceso de medicion de tiempo y memoria. Se consiguen las primeras mediciones coherentes con las expectativas, e investigo la manera de resumir los datos para la presentación de la memoria. Se opta por utilizar librerías de Python para el procesamiento y presentación de dichos datos.

1 de Junio - 21 de Junio Se establecen mecanismos que permitan la supervisión de la memoria del TFG en tiempo real, para agilizar su redacción. Los tutores instruyen y aconsejan sobre el maquetado, estilo, organización, metodología y presentación de la memoria en formato LaTeX. Para ello, el intercambio de información y las reuniones son constantes, y el ritmo de trabajo se agiliza, debido a la poca experiencia en la redacción de memorias técnicas por mi parte. Se ultiman las correcciones y preparaciones para la entrega del conjunto de materiales que constituyen este TFG.

Bibliografía

[1] V.S. Andrey Nazarenko Yurasov. *Space Surveillance Analysis*. Inf. téc. Report prepared by the Scientific-Industrial Firm "NUCLON" for the Charles Stark Draper Laboratory. [See Topic 5.2 Comparison of Satellite Theories for a detailed description of the USM Theory]. (document approved for public release). Jul. de 1999.

[2] Paul J. Cefola y col. «Comparison of the DSST and the USM semi-analytical orbit propagators». En: *Advances in the Astronautical Sciences* 114 (2003). Paper AAS 03-236, págs. 1943-1984.

[3] Mauro Massari y col. «Differential Algebra software library with automatic code generation for space embedded applications». En: *2018 AIAA Information Systems-AIAA Infotech @ Aerospace*. DOI: 10.2514/6.2018-0398. eprint: https://arc.aiaa.org/doi/pdf/10.2514/6.2018-0398. URL: https://arc.aiaa.org/doi/abs/10.2514/6.2018-0398.

[4] *Orbital elements*. Mayo de 2020. URL: https://en.wikipedia.org/wiki/Orbital_elements.

[5] E. Yurasov y col. «Universal Semianalytic Satellite Motion Propagation Method». En: *Space surveillance, AD REPORTS -NTIS- AD A, U.S.-Russian workshop; 2nd, Space surveillance*. NTIS, 1996, págs. 198-211. URL: https://www.tib.eu/de/suchen/id/BLCP%3ACN020676740.

[6] V.S. Yurasov. «The application of universal semi-analytical method for satellite motion propagation in the atmosphere». En: *Observation of artificial celestial objects* 82 (1984). USSR Academy of Sciences Astrocouncil, Moscow.